

Leveraging Flawed Tutorials for Seeding Large-Scale Web Vulnerability Discovery

Tommi Unruh¹, Bhargava Shastry¹, Malte Skoruppa², Federico Maggi³, Konrad Rieck⁴,
Jean-Pierre Seifert¹ and Fabian Yamaguchi⁴

¹TU Berlin, Berlin, Germany

²Saarland University, Saarland, Germany

³FTR, Trend Micro, Inc., Milan, Italy

⁴TU Braunschweig, Braunschweig, Germany

Abstract

The Web is replete with tutorial-style content on how to accomplish programming tasks. Unfortunately, even top-ranked tutorials suffer from severe security vulnerabilities, such as cross-site scripting (XSS), and SQL injection (SQLi). Assuming that these tutorials influence real-world software development, we hypothesize that code snippets from popular tutorials can be used to bootstrap vulnerability discovery at scale. To validate our hypothesis, we propose a semi-automated approach to find recurring vulnerabilities starting from a handful of top-ranked tutorials that contain vulnerable code snippets. We evaluate our approach by performing an analysis of tens of thousands of open-source web applications to check if vulnerabilities originating in the selected tutorials recur. Our analysis framework has been running on a standard PC, analyzed 64,415 PHP codebases hosted on *GitHub* thus far, and found a total of 117 vulnerabilities that have a strong syntactic similarity to vulnerable code snippets present in popular tutorials. In addition to shedding light on the anecdotal belief that programmers reuse web tutorial code in an ad hoc manner, our study finds disconcerting evidence of insufficiently reviewed tutorials compromising the security of open-source projects. Moreover, our findings testify to the feasibility of large-scale vulnerability discovery using poorly written tutorials as a starting point.

1 Introduction

Programming aids such as web tutorials and Q&A websites are popular with novice and expert programmers alike. To what extent, and how these aids influence the quality of real-world software remains an open question. On the one hand, popular Q&A websites, such as *stackoverflow.com*, have an in-built reputation system where correct advice gets up-voted through a consensus. Hence, a common expectation is that community-driven websites weed out bad coding suggestions.

On the other hand, the Web is replete with tutorial-style webpages that simply present curated snippets of code that accomplish a task. Most tutorials omit a discussion about API quirks or the security-impact a code snippet might have. It is not surprising, therefore, that the presented code snippets suffer from basic security vulnerabilities.

The connection between tutorials and vulnerabilities in real-world code is largely unknown. Although previous studies [9, 16, 29, 36] have shown that copy-pasted code can lead to recurring vulnerabilities, these studies have only considered instances of copy-pasted code *within* a codebase. We seek to generalize this result by asserting that, like code snippets originating in the same codebase, popular programming resources on the Web constitute a major source of documentation that is regularly consulted by developers and often introduces vulnerabilities into software.

Based on our assertion, we hypothesize that vulnerability discovery can be *seeded* by code snippets such as those found in top-ranked tutorials. Viewed from an adversarial standpoint, we present a novel approach for bootstrapping vulnerability discovery at scale. Our main intuition is that recurring vulnerabilities can be found by recognizing, and subsequently looking for patterns in code that correspond to the original vulnerability. We refer to instances of these patterns as *code analogues* throughout the rest of the paper. Our expectation is that if such a pattern recurs, so will the corresponding vulnerability. To identify code analogues, we automatically generate *graph traversals*, which can be used to mine code for these analogues using graph databases. Each graph traversal is derived from normalized fragments of a code snippet’s abstract syntax tree (AST), augmented with data-flow information. These graph traversals thus express syntactic properties of the original tutorial code.

Our workflow for finding recurring vulnerabilities consists of two steps. First, we automatically translate vulnerable code snippets into graph traversals, which

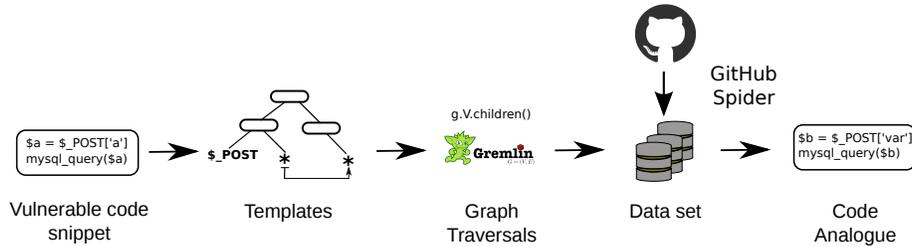


Figure 1: Workflow for finding recurring vulnerabilities.

concretize our expectations of code that constitutes an analogue. Second, we serialize a codebase under analysis into a graph database and employ the automatically generated traversals to search the database for analogue occurrences. As a result, we obtain a set of locations in application source code that bear a strong syntactic resemblance to vulnerable code snippets.

An empirical evaluation of our approach, on a data set of 64,415 web applications, shows that an adversary with access to a standard PC and a DSL broadband connection can leverage our techniques to efficiently discover recurring vulnerabilities in web application code (Section 3). Although AST is a fundamental construct for any programming language, we prototype our analysis framework for analyzing applications written in PHP, the most widely deployed server side scripting language to date [33]. Our analysis framework accepts vulnerable (tutorial) code snippets as input, and returns its analogues in a target PHP codebase. Since our analysis can not guarantee that the returned analogues are also vulnerable, we manually review them. Manual review has also been useful in teasing out the connection insecure tutorials and web application vulnerabilities. Thanks to our framework, we have uncovered over 100 vulnerabilities in web application code that bear a strong resemblance to vulnerable code patterns found in popular tutorials. More alarmingly, we have confirmed that 8 instances of a SQLi vulnerability present in different web applications are an outcome of code copied from a single vulnerable tutorial. Our results indicate that there is a substantial, if not causal, link between insecure tutorials and web application vulnerabilities.

In summary, we make the following contributions:

- We present a novel approach for bootstrapping large-scale vulnerability discovery, namely, leveraging flawed tutorial code to seed vulnerability search in application code. We evaluate this approach using a proof-of-concept framework that finds recurring vulnerabilities in PHP application code hosted on GitHub.
- We propose a computationally efficient method to search for recurring vulnerabilities. We translate

vulnerable snippets of code into graph traversals to identify code analogues in a program abstraction.

- We show that large-scale vulnerability scanning of publicly available open-source repositories is feasible, even with limited resources such as a standard PC and a broadband DSL connection.
- Finally, our results give credence to the widely known anecdote that programmers copy and paste code from vulnerable tutorials. Our case study, involving 64,415 PHP projects hosted on GitHub, indicates that such ad hoc code re-use may endanger the security of software throughout the open-source landscape.

Our tools GithubSpider, and CADetector are available at <https://github.com/tommiu/GithubSpider> and <https://github.com/tommiu/ccdetection> respectively.

2 Methodology

In this study, we take on an attacker’s perspective and design our method such that it allows for semi-automated discovery of recurring vulnerabilities without requiring special access to a hosting platform or considerable computational resources. Figure 1 illustrates our workflow, that we briefly describe in the following paragraphs.

1. **Derivation of templates from tutorials.** We extract templates from vulnerable code snippets contained in tutorials. These templates represent syntactical properties of the vulnerable code as well as information about data flow (Section 2.1).
2. **Generating traversals from templates.** We leverage templates to automatically generate *traversals* for a graph-based code mining system. Graph traversals enable us to scan large amounts of code for analogues of these vulnerable snippets in a computationally efficient manner (Section 2.2).
3. **Spidering code repositories.** We automatically collect a large data set of open-source code bases

from a code hosting site, choosing GitHub as a representative case study (Section 2.3).

4. **Mining for vulnerabilities.** Leveraging our analysis framework, we automatically mine the code of our data set for instances of vulnerable tutorial snippets. We manually cross-check if matches returned by our analysis platform constitute vulnerabilities (Section 2.4).

In the remainder of this section, we describe each of these steps in greater detail providing background information where necessary.

2.1 Derivation of Templates from Tutorials

Exact copies of vulnerable code snippets can be found using string matching utilities such as the standard UNIX tool `grep`. However, as programmers copy and paste code from tutorials, they are likely to adapt it slightly, for example, by changing the names of variables. Therefore this naïve approach fails in all but the most simple cases. To account for slight modifications, we require a method that is robust enough to identify sequences of statements *similar* to those found in the tutorial in terms of the operations they carry out on their input variables. We refer to these re-occurrences of tutorial code as *code analogues*, or simply analogues.

An elegant approach to address this problem is to extract intermediate graph representations from code that represent syntax and data flow, and formulate syntactical and data flow properties of the code snippet in terms of traversals in these graphs [see 13, 30, 35]. These traversals are formulated such that they succeed when the code matches, and fail when it does not. Although these graph traversals can be formulated manually, in this work, we devise a two-step procedure to automatically generate them from vulnerable snippets of code, making it possible to directly search for these snippets without additional manual work.

The first step of our procedure is to generate a *template* that encodes syntax and data flow of the code snippet that we attempt to scan for. To illustrate this process, we consider the vulnerable code snippet shown in Figure 2 taken from a popular PHP tutorial. The code contains one SQLi, and one stored XSS vulnerability. The SQLi vulnerability occurs on line 6 as the attacker-controlled POST-variable `$title` is used in the construction of an SQL query without first undergoing sanitization. The XSS vulnerability can be triggered on line 12, and 13 where databases rows are inserted into the document without escaping.

Two queries can be generated from the code snippet shown in Figure 2: one to identify instances of the SQLi vulnerability, and another for the XSS vulnerability. In

```
1 <?php
2 include "db.php";
3
4 $title=$_POST["title"];
5 $result=mysql_query("SELECT * FROM wp_posts where;
6     post_title like '%$title%'
7     and post_status='publish'");
8 $found=mysql_num_rows($result);
9
10 if($found>0){
11     while($row=mysql_fetch_array($result)){
12         echo "<li>$row[post_title]</br>
13             <a href=$row[guid]>$row[guid]</a></li>";
14     }
15 }else{
16     echo "<li>No Tutorial Found<li>";
17 }
18 // ajax search
19 ?>
```

Figure 2: Identified vulnerable tutorial, allowing for SQLi (line 6), and XSS (line 11-12).

the following walk-through, we focus on the SQLi vulnerability, as highlighted in Figure 2.

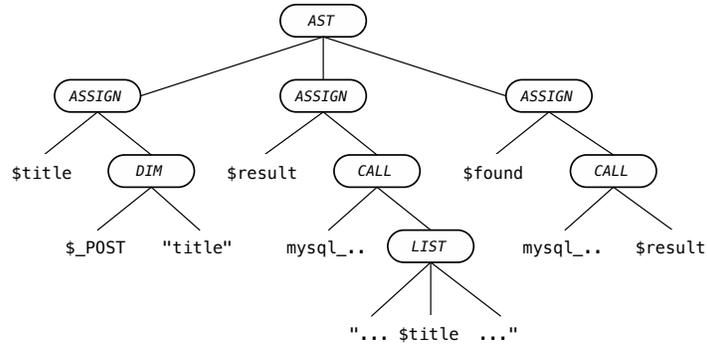
We proceed to generate an AST of the vulnerable code snippet, a standard tree-representation of program syntax. ASTs provide a hierarchical decomposition of code into its language elements. As an example, Figure 3(a) shows the abstract syntax tree for the SQLi vulnerability. In this tree, leaf nodes correspond to identifiers (e.g., `$title`), API symbols (e.g., `mysql...`), or literals (e.g., `SELECT`), and inner nodes represent operations such as assignments, function calls, or array indexing operations.

To derive a template from an AST, we replace all variables and literals by wildcard symbols and introduce edges between nodes representing the same variable. The template thus abstracts from concrete variable names and strings, while preserving data flow between variables. Figure 3(b) shows the corresponding template for our running example. As indicated by the data flow edges, the template enforces that there is a match if and only if the variable occurring on the left-hand side of the assignment in the first statement, and the variable appearing as an argument in a function call in the next statement are the same. An assignment to one variable, followed by the use of a different variable in a function call does not trigger a match, because there is no data flow between them.

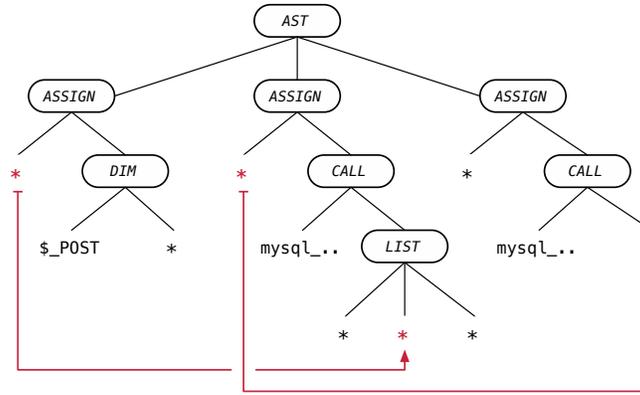
With templates for code snippets at hand, we are now ready to generate graph traversals that allow code to be mined for instances (analogues) of these snippets.

2.2 Generation of Traversals from Templates

Upon successful generation of a template for a vulnerable code snippet, we transform the template into a corresponding graph traversal. Although in principle, graph



(a) Abstract Syntax Tree.



(b) Derived Template.

Figure 3: Abstract syntax tree and derived template for the example in Figure 2.

traversals can be formulated in any programming language, the query language Gremlin [7] is specifically designed for this purpose. Moreover, traversals formulated in Gremlin can be executed on any graph database system that supports the Blueprints standard [6], a compatibility layer for graph databases similar to JDBC for relational databases.

For a given template, we generate a traversal that identifies ASTs with (1) the same node types, (2) in the same order, and (3) nested in the same way. Moreover, the traversal succeeds only if data flow between statements is in correspondence with the data-flow edges of the template.

In essence, the AST structure is encoded by formulating chains of filter operations that succeed only if the desired node types can be matched. To account for data flow, the names of variables are stored as the AST is traversed, and filtering is performed to ensure that the correct variable names occur throughout the tree. Algorithm 1 describes this process in detail. Each child node

of the template is converted into a traversal via the recursive function `ConvertNode`, and `prepareNode` adds code to traverse to the next child node.

As an example, Figure 4 shows the traversal generated by this procedure for the vulnerable code snippet in Figure 2. For all nodes in the AST, the traversal attempts to match the left subtree, the subtree in the middle, and the right subtree starting at lines 4, 20, and 37 respectively. Subtrees are matched by applying chains of filter expressions. For instance, to match the left-most subtree consisting of an assignment with a variable as a child node, the traversal first attempts to match an assignment, and, on success, determines whether the child node is a variable node using a subsequent filter (see lines 4 and 5). Finally, lightweight data-flow tracking is implemented by storing the names of variables as they are first encountered (e.g., on line 8), and filtering based on these names (e.g., line 38).

The generated traversals implement a computationally efficient code analogue detection. For classic code

Algorithm 1 Pseudocode for translating templates into graph traversals.

```

1: function TRAVERSAL(template)
2:    $t = \emptyset$ 
3:   for each child of template do
4:      $t += \text{convertNode}(\textit{child})$ 
5:      $t += \text{prepareNextSubtree}()$ 
6:   return  $t$ 
7: function CONVERTNODE(child)
8:    $t = \emptyset$ 
9:   ▷ Ensure correct node-type
10:   $t += \text{filter}\{ \text{"node.type"} == \textit{child.type} \}$ 
11:  if  $\text{node.type} == \text{"VAR"}$  then
12:    if variable with name has been used before
13:    then
14:       $t += \text{filter}\{ \text{"this\_node's name"} == \textit{name} \}$ 
15:    else
16:       $t += \text{sideEffect}\{ \text{Remember the variable} \}$ 
17:    return  $t$ 
18:  for each child\_node of child do
19:     $t += \text{convertNode}(\textit{child\_node})$ 
20:  return  $t$ 
21: function PREPARENEXTSUBTREE
22:  return  $\{ \text{traverse next child} \}$ 

```

clone detection tools, the comparison of every subtree of a given AST of N nodes with every subtree of the same AST costs $O(N^3)$ and can become $O(N^4)$ when also comparing sequences of trees. Since empirically, a large software system of M lines of code has $N = 10M$ AST nodes, this AST-based approach suffers a hard scaling problem. There is a solution to this runtime cost, which involves comparing hashes of AST subtrees instead of the AST subtrees themselves. Then, the runtime becomes $O(N)$ [see 3]. However, hashing subtrees will not work with our approach.

In our approach, we are comparing a given source AST S of N nodes, obtained from a tutorial, with a target AST T of M nodes, which is the code base that should be scanned for code analogues of S . Therefore, N is much smaller than M . For this approach, the runtime is much smaller. More formally, our algorithm starts out with comparing the first node in N with every node in M , until it finds one that is equal. This costs $O(M)$. Then, it will compare the other nodes as well, until all nodes in S were matched successfully in T . Therefore, when a code analogue is found, we did N comparisons to match it. For subtrees that are not analogues of S , we do 1 to $N - 1$ comparisons. This cost is $O(N)$. In the worst case, this entails a full check that costs $O(N)$ for every node

```

1  g.V.sideEffect{ childnumber = 0 }
2
3  // Left subtree
4  .filter{ isType(it, "ASSIGN") }
5  .filter{ isType(it.ithChildren(0), "VAR") }
6
7  // Remember variable "title"
8  .sideEffect{ _title =
9    it.ithChildren(0).varToName().next() }
10 .filter{ isType(it.ithChildren(1), "DIM") }
11 .filter{ isType(it.ithChildren(1).ithChildren(0),
12   "VAR") }
13 .sideEffect{ childnumber = it.childnum }
14 .sideEffect{ childnumber = childnumber + 1 }
15
16 .parents().children().filter{
17   it.childnum == childnumber }
18
19 // Subtree in the middle
20 .filter{ isType(it, "ASSIGN") }
21 .filter{ isType(it.ithChildren(0), "VAR") }
22
23 // Remember variable "result"
24 .sideEffect{ _result =
25   it.ithChildren(0).varToName().next() }
26 .filter{ isType(it.ithChildren(1), "CALL") }
27 .filter{ isType(it.ithChildren(1)
28   .ithChildren(0), "NAME") }
29 .filter{ isType(it.ithChildren(1)
30   .ithChildren(1), "ARG_LIST") }
31 ...
32
33 .sideEffect{ childnumber = it.childnum }
34 .sideEffect{ childnumber = childnumber + 1 }
35
36 // Right subtree
37 .parents().children()
38 .filter{ it.childnum == childnumber }
39 .filter{ isType(it, "ASSIGN") }
40 .filter{ isType(it.ithChildren(0), "VAR") }
41
42 // Remember variable "found"
43 .sideEffect{ _found =
44   it.ithChildren(0).varToName().next() }
45 .filter{ isType(it.ithChildren(1), "CALL") }
46 .filter{ isType(
47   it.ithChildren(1).ithChildren(0), "NAME") }
48 .filter{ isType(it.ithChildren(1)
49   .ithChildren(1), "ARG_LIST") }
50 ...

```

Figure 4: Generated traversal for the code snippet in Figure 2.

in T . Thus, the complexity of our approach is $O(NM)$, or for cases where $N \ll M$ holds, $O(M)$.

Our algorithm can benefit from knowledge about the depth of nodes. Let $d(x)$ denote the depth of a node x , then, comparisons beyond an AST depth of $d(T) - d(S) + 1$ can be safely avoided, given that nodes at the depth cannot contain the AST S . However, we typically make use of small vulnerable code snippets ($N \ll M$), for which the saving in computational cost is marginal.

Translation of tutorial templates into graph traversals provides us a means to mine large amounts of open-source code for recurring vulnerabilities. Since we assume that the attacker is not an insider (who can stealthily plug our analyses into the back-end of a hosting platform), analysis impinges on the attacker being able to download a project's source code. To this end,

we have implemented a tool called *GithubSpider* to facilitate code crawling at scale. In the next section, we briefly discuss its design and implementation.

2.3 Spidering Code Repositories

We make a conservative estimate of computing resources that an attacker has at his/her disposal. We assume that an attacker has access to a modest computing device such as a standard PC, and a DSL broadband connection. Our threat model lowers the barrier to entry for performing vulnerability discovery at scale. Additionally, we assume that the attacker makes every effort to be stealthy. In the context of code spidering, this implies that rate limits imposed by a hosting service cannot be abused. Thus, we are constrained to spider at a modest speed. The constraints that we impose on the attacker make our tooling and analysis operate in a real-world setting. Understanding whether an attacker, in spite of these constraints, can use our techniques to find vulnerabilities in open-source code is part of our research question.

Although multiple code hosting platforms provide APIs that could be used for spidering, we focus on GitHub in our case study. GitHub is the most popular open-source collaboration platform, hosting approximately 10.3 million developers, and over 25 million projects [10]. In addition to code, GitHub maintains meta-data such as a project’s language, creation time, popularity (*stars* and *forks*), and total code size.

Our spidering tool, *GithubSpider*, uses GitHub’s REST API to obtain project meta-data for a large number of projects. *GithubSpider* is designed to be general purpose: It obtains project meta-data and applies user-defined filters to them, providing the flexibility required to spider code repository classes of interest. Thus, *GithubSpider* can be leveraged towards analysis of a different family of codebases, say *C/C++* code. We leverage *GithubSpider* to crawl projects written in the PHP language. Additionally, we filter projects based on their popularity. We gauge a project’s popularity by the number of stars¹ it has received. After narrowing down projects of interest, we leverage *GitHubspider* to clone (download) projects and their revision history (see Section 3.3).

2.4 Mining for Vulnerabilities

Finally, we automatically mine the downloaded code repositories using the traversals generated from vulnerable code snippets (see Section 2.2). We achieve this by importing the code into the code-mining platform *Joern* [35]. *Joern* first parses source files to generate ASTs, and subsequently imports the ASTs into a Neo4j graph

¹A GitHub user can express interest in a repository by starring it.

Algorithm 2 Algorithm to find code analogues using graph traversals.

```
1: procedure MINECODE(repositories, queries)
2:   for each repository in repositories do
3:     create ASTs for repository
4:     import ASTs into graph database and start
       server
5:     for each query in queries do
6:       run query against database server
7:       save statistical data
8:       save code clone data
9:     stop graph database server
```

database, which allows us to efficiently execute graph traversals and collect statistical data about our analysis.

Scanning the large number of projects obtained from our spider in an acceptable time frame is challenging task, as graph matching needs to be performed on all ASTs of all retrieved code bases. Fortunately, projects can be processed independently, and therefore, we can distribute scanning across several processes running in parallel by splitting spidered repositories into groups. This task can also be carried out in a distributed setting.

For a given set of queries and a group of repositories, Algorithm 2 summarizes the scanning process that we carry out. For each repository, the algorithm creates ASTs for each of its source files and imports them into the graph database. The graph database server is subsequently started, each query is executed against the databases, and statistical data such as query execution time is collected. Upon running all queries on a project, the graph database server is stopped.

We have implemented the presented algorithm in a tool, that we call *CADetector*, short for code analogue detector. *CADetector* takes automatically generated queries (Gremlin traversals), and a PHP project for analysis as input, and returns matching code as output. Code analogues are presented to a human analyst for review. In the next section, we present our evaluation methodology and results.

3 Evaluation

Our evaluation follows a two-step process. First, we select a handful of vulnerable code snippets, obtained from top-ranked PHP tutorials, to seed our study. Second, if we find a match for a vulnerable snippet in our data set as a result of our analysis, we flag it for manual review. Based on this two-step process, we have performed extensive evaluation of our analysis framework. Our analysis data set consists of 64,415 PHP codebases that have been downloaded using *GithubSpider*. To gauge its fea-

```

1 <?php
2 // Check for existing user with the new id
3
4 $sql = "SELECT COUNT(*) FROM user WHERE
5         userid = '$_POST[newid]'";
6 $result = mysql_query($sql);
7 if (!$result) {
8     error('A database error occurred in
9     processing your submission.\nIf this
10    error persists, please contact
11    you<at>example.com.');
```

Figure 5: Vulnerable code snippet from tutorial, containing a SQL injection vulnerability between lines 4–6 (slice).

sibility, we have run our analysis against the top 10 PHP codebases on GitHub, in addition to the codebases in our data set. The following paragraphs describe how seeds were obtained and queries generated (Section 3.1), the nature of the analysis data set (3.3), and our analysis results (3.4).

3.1 Identification of Vulnerable Tutorials

To identify widely read tutorials, we query the Google search engine using the following set of terms.

```

"mysql tutorial"
"php database user"
"php mysql user query"
"php search form"
"php ajax search tutorial"
"javascript echo user input"
```

For each of these search terms, we manually review the first five results returned by the search engine. We evaluate each of the tutorials for SQLi and XSS vulnerabilities by following established secure programming guidelines by the Open Web Application Security Project (OWASP), namely, the guide on *Reviewing Code for SQL Injection* [14], and the *Cross Site Scripting Prevention Cheat Sheet* [28]. Among the top five results (30 in total), we find 9 tutorials that contain vulnerable code: 6 tutorials with SQLi, and 3 tutorials with XSS. A snippet from a representative tutorial containing both SQLi and XSS vulnerabilities is shown in Figure 2.

3.2 Query Generation

We expect copy-pasted code from vulnerable tutorials to result in recurring vulnerabilities in application code. A more subtle manner in which recurring vulnerabilities may manifest themselves is when developers follow

the same (vulnerable) programming idiom(s) presented by a tutorial. To cater to both possibilities, we generate queries using an exact replica of code present in a tutorial (normal query), and a slice of the tutorial code containing the vulnerability (strict query). A strict query abstracts only the vulnerable slice of code, whereas a normal query abstracts the entire tutorial. We use strict queries to identify known vulnerable patterns in web applications, and normal queries to identify code analogues of tutorial code. Figure 5 illustrates the difference between code snippets used for generating normal, and strict queries. Entire tutorial code is shown in the listing with the vulnerable slice highlighted in red. Lines 4–6 of the tutorial contain a classic SQLi vulnerability: Unsanitized user-input from a POST variable is used in a MySQL query. The vulnerable slice contains only these lines, which represents the minimal working snippet containing the vulnerability.

3.3 Analysis Data-set

We leverage GitHub to obtain a large data set of web applications for analysis. GitHub hosts over 25 million projects written in several programming languages. Thus, we filter content (PHP projects) that is relevant to us. Our crawler inspects project metadata (accessible via REST API) to perform the filtering. GithubSpider, our code crawler implementation, has filtered through a total of 462,069 PHP repositories on GitHub. Of these, we have downloaded a total of 64,415 PHP codebases for analysis. These codebases comprise our analysis data set.

We divide our data set into three groups by popularity. We quantify the popularity of a codebase by the number of times it has been *starred* by users on GitHub. Our classification results in the following data set partitions:

1. **Barely known projects (Not popular).** 42,064 projects that are starred at most three times and have a total file size of less than 3 MB.
2. **Projects known by several people (Popular).** 16,037 projects that are starred at least four times, but at most nine times, and have a total file size of less than 3 MB.
3. **Popular projects (Very popular).** 6,314 projects that are starred at least ten times and have a total file size of less than 3 MB.

GitHub imposes a rate limit of 5000 API requests per authenticated user per hour. The imposed rate limit proved to be the main bottleneck in downloading repositories. Although using multiple authenticated accounts for crawling is a simple workaround for removing the bottleneck, we stayed clear of it.

Data set	Size	Code analogues	Vulnerabilities (percentage)
Not popular	42,064	269	80 (29.74%)
Popular	16,037	528	35 (6.63%)
Very popular	6,314	23	2 (8.7%)
Total	64,415	820	117 (14.27%)

Table 1: Analysis summary for codebases in our data set. In total, 820 code analogues were found which included 117 recurring vulnerabilities. The table shows a break down of our findings in each data set partition.

3.4 Analysis Results

We used auto-generated graph traversals (queries) to mine for analogues in our analysis data set. Discovered analogues were manually reviewed, and vulnerable analogues identified. Table 1 shows an overview of the analogues, and vulnerabilities discovered by our analyses. With under two dozen graph traversals generated from a handful of vulnerable tutorials, we obtained 820 code analogues of which 117 were found to be vulnerable. We found that string normalization in the AST resulted in non-exact matches. For instance, a query generated from the code snippet `$var = $_GET['var']` matched not only its replica but also a seemingly benign snippet such as `$var = $value['id']`. Thus, matches returned by the traversals had to be manually validated. In spite of string normalization, we found that automatically generated queries result in interesting corner cases. Among non popular codebases, roughly 1 out of 3 code analogues is vulnerable, and on average 1 out of 7 analogues is vulnerable across the entire data set. Analogues are localized to a small portion of application code, which facilitates manual review of all candidates returned by our analyzer.

Newly discovered vulnerabilities We manually verified a total of 117 vulnerabilities in our data set. Of these, 8 vulnerabilities were replicas of code from a popular SQL tutorial that we found on the first Google results page. Although all of the 8 vulnerabilities were found among non popular code repositories, the finding shows that ad-hoc code reuse is a reality. We are in the process of notifying the tutorial authors about our findings. Our hope is that the presented vulnerabilities are fixed in a timely manner, so that developers borrowing code from these tutorials in the future will not inherit the same vulnerabilities in their code.

```

1 // SQL injection vulnerability
2 $questionQuery = "SELECT * FROM Questions
3     WHERE AssnID='$_GET[AssnID]'";
4 $questionResult = mysql_query($questionQuery);

1 // Path-traversal vulnerability
2 $filename = "signatures/{$_GET['id']}.png";
3 $handle = fopen($filename, "r");

1 // Potential SQL injection vulnerability
2 $oldBusSeats = "SELECT businessseats FROM flight
3     WHERE flightNo = $_flight[0]";
4 $busseats = mysql_query($oldBusSeats);

```

Figure 6: The figure shows three analogues returned by CADetector that match the traversal generated for the vulnerable slice in Figure 5. Although we expected to find only SQLi vulnerabilities, we also uncovered a path-traversal vulnerability that matched our query template.

80% of the discovered vulnerabilities were SQLi vulnerabilities, and the rest were XSS, and path-traversal vulnerabilities. As shown in Table 1, the proportion of vulnerable codebases is higher among low popularity codebases, compared to medium and high popularity codebases. During manual review, we found a plausible explanation for this disparity in vulnerability density: PHP applications in the moderate and high popularity categories make consistent use of newer, and more secure, MySQL APIs in PHP which are not vulnerable to classic (first-order) SQLi attacks. We found that use of the PDO_MySQL interface [31], and the MySQLi extension [32] was widespread among these codebases.

3.5 Discussion

Manual review of the code analogues and vulnerabilities returned by our analysis framework suggests that, graph traversals are very good at eliciting vulnerable snippets in a large amount of code. Firstly, our approach ensures that the code analogues that we find are small snippets of code, typically spanning under 10 lines of code in our dataset. Figure 6 shows three code analogues returned by CADetector for a query originating from the vulnerable code snippet in Figure 5. All three analogues span two lines of code and can be quickly assessed by a human analyst.

Secondly, we find that the abstraction that we choose (AST augmented with data-flow information) is robust. For example, we discovered a path-traversal vulnerability in the process of mining a codebase using a SQLi query. Indeed, both vulnerabilities share the same syntactic structure of code: A tainted PHP variable is used in a security-sensitive PHP function call (`mysql_query` and `fopen`). However, since our abstraction does not convey information about taintedness of data, our analysis returns analogues where vulnerabilities may need to be

manually verified. The third analogue shown in Figure 6 serves as a demonstrative example. The analogue contains a potential SQLi vulnerability. It is vulnerable if the PHP variable `$flight` reads from a tainted variable (say, an attacker-controlled `$_POST` variable). This needs to be manually verified in our setup.

Analysis runtime Our timing measurements showed that our analysis is fast for even relatively large codebases. For the top 10 PHP code repositories on GitHub, CADetector analysis runtime varies between 19 seconds (for the *laravel* project, 777 lines of code) and 53 minutes (for *symfony*, 209 thousand lines of code). Our evaluations suggest that CADetector is a fast analogue detector for codebases with hundreds of thousands of lines of code and above.

4 Related Work

Our work touches upon two distinct problems: finding similar code, and flagging vulnerabilities in source code. In the following, we contextualize our work in both domains.

Code clone detection Despite modern software design processes and state of the art programming environments, real-world software development accommodates ad-hoc code re-use. In their seminal work on code clone detection, Baxter et al. [3], citing earlier work [1, 25], state that 5-10% of source code is duplicated in large software projects. The initial motivation for code clone detection was that ridding software of seemingly redundant code might achieve a performance gain. Thus, traditional code clone detection tools seek code replicas in a single codebase, or a set of codebases with the same provenance. This has guided the design of several code clone detection tools [2–5, 8, 15, 17, 18, 20, 21, 21–24, 26, 27].

Recent research [16, 29] has shown that code clones pose a more serious threat: Vulnerabilities in cloned code get propagated but their fixes do not. ReDeBug [16] flags unpatched code clones by finding replicas of a known vulnerable snippet in an OS distribution. Like earlier proposals on code clone detection, ReDeBug flags clones within codebases of similar provenance, because of which it may look for exact matches. In contrast, we cannot always expect to find borrowed code from an external source *as is*: Developers typically adapt tutorial code for their own end. This subtle difference precludes the use of hashing functions to measure similarity of code in our work. Instead, our queries attempt to recognize the structure of vulnerable code.

Yamaguchi et al. [36] propose a machine-learning based method for *extrapolating* (i.e., finding other instances of) known vulnerable code patterns that are manually specified. Our work is closer to theirs in that we employ structural code fragments (such as AST fragments) to drive the search for vulnerabilities. Having said that, a notable difference is that Yamaguchi et al. perform computations on a code abstraction (specifically, a vector space). In our work, the query for a similar code snippet is concretized in the form of graph traversals. Moreover, we automatically generate vulnerability patterns from code snippets.

Vulnerability discovery Since we use static program analysis in discovering vulnerabilities, we shall restrict our discussion to prior work in this domain. The dynamic nature of web programming languages, such as PHP and JavaScript, has made static analysis of web applications a challenging task. Researchers have approached vulnerability discovery in PHP code as a static taint analysis problem: Detect the flow of untrusted user input into a security sensitive sink. Pixy [19] is a static analysis tool that flags XSS, and SQLi vulnerabilities in PHP codebases. In the same vein as Pixy, Xie et al. [34] present a summary-based static analysis algorithm to discover security vulnerabilities in PHP code. Our proposal is not *another* vulnerability scanner for PHP code. Rather, our techniques provide a means to draw inferences about unsafe coding practices among web application developers. Considering that web applications are ultimately user-facing programs that handle sensitive data, our study is timely.

5 Limitations and Future Work

A limitation of our study is that our prototype restricts the evaluation scope of recurring vulnerability detection to PHP application code; that is, we cannot say that programmers employing other languages are similarly prone to copying from tutorials. Moreover, we restrict our analysis to open-source code, and thus, the possibility exists that the practice of copying from tutorials is particularly prevalent in the open-source world and less common in closed-source environments. Exploring these questions is left for future work.

For the detection of code analogues, we employ an approach that allows the names of identifiers to be changed, but is otherwise strict about the code it matches. For example, if additional statements are introduced in between statements of a seed, we do not detect the corresponding code as a clone. This is a deliberate design choice. Although it may result in the discovery of fewer tutorials, the identified code is more similar to that contained in the

tutorial, and therefore, more likely to have been copied from it.

Our *approach*—formulating graph traversal queries from code snippets, and issuing these queries in a code mining system—is generic enough to be decoupled from the specifics of a programming language. Thus, our analysis techniques can be incorporated into existing code analysis platforms such as Kythe [11], Joern [35], and Frappé [12]. Systematically leveraging popular tutorial content from the Web to seed vulnerability discovery is an avenue for future research. For instance, portals such as Google Trends can be queried to obtain high-value seeds for vulnerability discovery.

6 Conclusion

Developers routinely consult programming resources as software is written. Although formal documentation such as language and API reference manuals provide detailed guidance, tutorials on the Web are as easily available and are more succinct. The lure of quick actionable advice makes tutorials an appealing reference for developers. We find that tutorials are not only ubiquitous on the Web but also very popular, consistently appearing in the first Google results page. Several tutorials betray a lack of understanding of secure coding practices advocated by well-regarded online communities such as OWASP. In our large-scale case study, we find over 100 vulnerable code snippets in application code that are syntactically similar, and in 8 instances identical, to tutorial code. These findings corroborate our hypothesis that vulnerable tutorials can be used to seed large-scale vulnerability discovery. They also suggest that there is a pressing need for code audit of widely consumed tutorials, perhaps with as much rigor as for production code.

We show that the syntactic structure of code can be used to infer similarities between code snippets of different provenance. Because syntactic analysis is relatively lightweight, it is fast enough to mine a large number of differently-sized codebases for recurring vulnerabilities. Our large-scale study is a testament to the efficiency of our proposal. This also means that there is low barrier to entry for performing vulnerability discovery at scale. Even an adversary with access to modest computing resources may be seen as direct threat to the security of software in the large open-source landscape. Although our study provides a single data point for an objective assessment of both our adversarial strategy and the connection between tutorials and real-world software, there appears to be promise in the applicability of our techniques to other application classes.

Acknowledgements. This work was supported by the Bundesministerium für Bildung und Forschung (BMBF) grant under Award No. KIS1DSD032 (Project Enzevalos). The opinions, views, and conclusions contained herein are those of the author(s) and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of BMBF.

References

- [1] Baker, B.S.: On finding duplication and near-duplication in large software systems. In: Reverse Engineering, 1995., Proceedings of 2nd Working Conference on. pp. 86–95. IEEE (1995)
- [2] Balazinska, M., Merlo, E., Dagenais, M., Lagüe, B., Kontogiannis, K.: Advanced clone-analysis to support object-oriented system refactoring. In: Reverse Engineering, 2000. Proceedings. Seventh Working Conference on. pp. 98–107. IEEE (2000)
- [3] Baxter, I.D., Yahin, A., Moura, L., Anna, M.S., Bier, L.: Clone detection using abstract syntax trees. In: Software Maintenance, 1998. Proceedings., International Conference on. pp. 368–377. IEEE (1998)
- [4] Bulychev, P., Minea, M.: An evaluation of duplicate code detection using anti-unification. In: Proc. 3rd International Workshop on Software Clones. Citeseer (2009)
- [5] Ducasse, S., Rieger, M., Demeyer, S.: A language independent approach for detecting duplicated code. In: Software Maintenance, 1999.(ICSM’99) Proceedings. IEEE International Conference on. pp. 109–118. IEEE (1999)
- [6] Foundation, A.S.: Blueprints property graph model. <https://github.com/tinkerpop/blueprints/wiki/Property-Graph-Model>, online; accessed 21-January-2016
- [7] Foundation, A.S.: Gremlindocs. <http://gremlindocs.spmallette.documentup.com/>, online; accessed 21-January-2016
- [8] Gabel, M., Yang, J., Yu, Y., Goldszmidt, M., Su, Z.: Scalable and systematic detection of buggy inconsistencies in source code. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications. pp. 175–190. OOPSLA ’10, ACM, New York, NY, USA (2010), <http://doi.acm.org/10.1145/1869459.1869475>
- [9] Gauthier, F., Lavoie, T., Merlo, E.: Uncovering access control weaknesses and flaws with security-discordant software clones. In: Proceedings of the 29th Annual Computer Security Applications Conference. pp. 209–218. ACSAC ’13, ACM, New York, NY, USA (2013), <http://doi.acm.org/10.1145/2523649.2523650>
- [10] GitHub, Inc.: Github.com. <https://github.com/>, online; accessed 24-July-2015

- [11] Google Inc.: Kythe. <https://www.kythe.io/>
- [12] Hawes, N., Barham, Ben, O.L.A.: Frappé: Using clang to visualize large codebases. <http://llvm.org/devmtg/2014-10/Slides/Hawes-Frappe.pdf>
- [13] He, B., Rastogi, V., Cao, Y., Chen, Y., Venkatakrishnan, V., Yang, R., Zhang, Z.: Vetting ssl usage in applications with sslint. In: Proc. of the IEEE Symposium on Security and Privacy (S&P) (2015)
- [14] Herlea, T.: Reviewing code for sql injection. https://www.owasp.org/index.php?title=Reviewing_Code_for_SQL_Injection&oldid=88988, online; accessed 02-January-2016
- [15] Hummel, B., Juergens, E., Heinemann, L., Conradt, M.: Index-based code clone detection: incremental, distributed, scalable. In: Software Maintenance (ICSM), 2010 IEEE International Conference on. pp. 1–9. IEEE (2010)
- [16] Jang, J., Agrawal, A., Brumley, D.: Redebug: Finding unpatched code clones in entire os distributions. In: 2012 IEEE Symposium on Security and Privacy. pp. 48–62 (May 2012)
- [17] Jia, Y., Binkley, D., Harman, M., Krinke, J., Matsushita, M.: Kclone: a proposed approach to fast precise code clone detection. In: Third International Workshop on Detection of Software Clones (IWSC). Citeseer (2009)
- [18] Jiang, L., Mishserghi, G., Su, Z., Glondu, S.: Deckard: Scalable and accurate tree-based detection of code clones. In: Proceedings of the 29th International Conference on Software Engineering. pp. 96–105. ICSE '07, IEEE Computer Society, Washington, DC, USA (2007), <http://dx.doi.org/10.1109/ICSE.2007.30>
- [19] Jovanovic, N., Kruegel, C., Kirda, E.: Pixy: a static analysis tool for detecting web application vulnerabilities. In: 2006 IEEE Symposium on Security and Privacy (S P'06). pp. 6 pp.–263 (May 2006)
- [20] Juergens, E., Deissenboeck, F., Hummel, B.: Clonedetective-a workbench for clone detection research. In: Proceedings of the 31st International Conference on Software Engineering. pp. 603–606. IEEE Computer Society (2009)
- [21] Kamiya, T., Kusumoto, S., Inoue, K.: Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. Software Engineering, IEEE Transactions on 28(7), 654–670 (2002)
- [22] Komondoor, R., Horwitz, S.: Using slicing to identify duplication in source code. In: Static Analysis, pp. 40–56. Springer (2001)
- [23] Koschke, R., Falke, R., Frenzel, P.: Clone detection using abstract syntax suffix trees. In: Reverse Engineering, 2006. WCRE'06. 13th Working Conference on. pp. 253–262. IEEE (2006)
- [24] Krinke, J.: Identifying similar code with program dependence graphs. In: Reverse Engineering, 2001. Proceedings. Eighth Working Conference on. pp. 301–309. IEEE (2001)
- [25] Lague, B., Proulx, D., Mayrand, J., Merlo, E.M., Hudepohl, J.: Assessing the benefits of incorporating function clone detection in a development process. In: Software Maintenance, 1997. Proceedings., International Conference on. pp. 314–321. IEEE (1997)
- [26] Lanubile, F., Mallardo, T.: Finding function clones in web applications. In: Software Maintenance and Reengineering, 2003. Proceedings. Seventh European Conference on. pp. 379–386 (March 2003)
- [27] Li, Z., Lu, S., Myagmar, S., Zhou, Y.: Cp-miner: finding copy-paste and related bugs in large-scale software code. IEEE Transactions on Software Engineering 32(3), 176–192 (March 2006)
- [28] OWASP: Xss (cross site scripting) prevention cheat sheet. [https://www.owasp.org/index.php/XSS_\(Cross_Site_Scripting\)_Prevention_Cheat_Sheet#Why_Can.27t_I_Just_HTML_Entity_Encode_Untrusted_Data.3F](https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet#Why_Can.27t_I_Just_HTML_Entity_Encode_Untrusted_Data.3F), online; accessed 19-October-2015
- [29] Pham, N.H., Nguyen, T.T., Nguyen, H.A., Nguyen, T.N.: Detection of recurring software vulnerabilities. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering. pp. 447–456. ASE '10, ACM, New York, NY, USA (2010), <http://doi.acm.org/10.1145/1858996.1859089>
- [30] Reps, T.: Program analysis via graph reachability. Information and Software Technology (1998)
- [31] The PHP Documentation Group: Php: Mysql (pdo) manual. <https://secure.php.net/manual/en/ref.pdo-mysql.php>
- [32] The PHP Documentation Group: Php: Mysql manual. <https://secure.php.net/manual/en/book.mysql.php>
- [33] w3techs.com: Usage of server-side programming languages for websites. http://w3techs.com/technologies/overview/programming_language/all (2015), online; accessed 17-october-2015
- [34] Xie, Y., Aiken, A.: Static detection of security vulnerabilities in scripting languages. In: Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15. USENIX-SS'06, USENIX Association, Berkeley, CA, USA (2006), <http://dl.acm.org/citation.cfm?id=1267336.1267349>
- [35] Yamaguchi, F., Golde, N., Arp, D., Rieck, K.: Modeling and discovering vulnerabilities with code property graphs. In: Proc. of IEEE Symposium on Security and Privacy (S&P) (2014)

- [36] Yamaguchi, F., Lottmann, M., Rieck, K.: Generalized vulnerability extrapolation using abstract syntax trees. In: Proceedings of the 28th Annual Computer Security Applications Conference. pp. 359–368. ACM (2012)